

EXHIBIT E

G:\eclair21 - GOOGLE-00-00000525\dalvik\libcore\luni\src\main\java\java\util\ComparableTimSort.java

```

1  /*
2   * Copyright (C) 2008 The Android Open Source Project
3   *
4   * Licensed under the Apache License, Version 2.0 (the "License");
5   * you may not use this file except in compliance with the License.
6   * You may obtain a copy of the License at
7   *
8   *     http://www.apache.org/licenses/LICENSE-2.0
9   *
10  * Unless required by applicable law or agreed to in writing, software
11  * distributed under the License is distributed on an "AS IS" BASIS,
12  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
13  * implied.
14  * See the License for the specific language governing permissions and
15  * limitations under the License.
16  */
17 package java.util;
18
19 /**
20  * This is a near duplicate of {@link TimSort}, modified for use with
21  * arrays of objects that implement {@link Comparable}, instead of using
22  * explicit comparators.
23  *
24  * <p>If you are using an optimizing VM, you may find that
25  * ComparableTimSort
26  * offers no performance benefit over TimSort in conjunction with a
27  * comparator that simply returns {@code
28  * ((Comparable)first).compareTo(Second)}.
29  * If this is the case, you are better off deleting ComparableTimSort to
30  * eliminate the code duplication. (See Arrays.java for details.)
31  */
32 class ComparableTimSort {
33     /**
34      * This is the minimum sized sequence that will be merged.  Shorter
35      * sequences will be lengthened by calling binarySort.  If the
36      * entire
37      * array is less than this length, no merges will be performed.
38      *
39      * This constant should be a power of two.  It was 64 in Tim Peter's
40      * C
41      * implementation, but 32 was empirically determined to work better
42      * in
43      * this implementation.  In the unlikely event that you set this
44      * constant
45      * to be a number that's not a power of two, you'll need to change
46      * the
47      * {@link #minRunLength} computation.
48      *
49      * If you decrease this constant, you must change the stackLen
50      * computation in the TimSort constructor, or you risk an
51      * ArrayOutOfBounds exception.  See listsort.txt for a discussion
52      * of the minimum stack length required as a function of the length
53      * of the array being sorted and the minimum merge sequence length.
54      */
55     private static final int MIN_MERGE = 32;
56
57     /**
58      * The array being sorted.
59      */

```

UNITED STATES DISTRICT COURT
NORTHERN DISTRICT OF CALIFORNIA

TRIAL EXHIBIT 45.2

CASE NO. 10-03561 WHA

DATE ENTERED _____

BY _____

DEPUTY CLERK

G:\eclair21 - GOOGLE-00-00000525\dalvik\libcore\luni\src\main\java\java\util\ComparableTimSort.java

```

53     private final Object[] a;
54
55     /**
56      * When we get into galloping mode, we stay there until both runs
57      * win less
58      * often than MIN_GALLOP consecutive times.
59      */
60     private static final int MIN_GALLOP = 7;
61
62     /**
63      * This controls when we get *into* galloping mode. It is
64      * initialized
65      * to MIN_GALLOP. The mergeLo and mergeHi methods nudge it higher
66      * for
67      * random data, and lower for highly structured data.
68      */
69     private int minGallop = MIN_GALLOP;
70
71     /**
72      * Maximum initial size of tmp array, which is used for merging.
73      * The array
74      * can grow to accommodate demand.
75      *
76      * Unlike Tim's original C version, we do not allocate this much
77      * storage
78      * when sorting smaller arrays. This change was required for
79      * performance.
80      */
81     private static final int INITIAL_TMP_STORAGE_LENGTH = 256;
82
83     /**
84      * Temp storage for merges.
85      */
86     private Object[] tmp;
87
88     /**
89      * A stack of pending runs yet to be merged. Run i starts at
90      * address base[i] and extends for len[i] elements. It's always
91      * true (so long as the indices are in bounds) that:
92      *
93      *     runBase[i] + runLen[i] == runBase[i + 1]
94      *
95      * so we could cut the storage for this, but it's a minor amount,
96      * and keeping all the info explicit simplifies the code.
97      */
98     private int stackSize = 0; // Number of pending runs on stack
99     private final int[] runBase;
100     private final int[] runLen;
101
102     /**
103      * Asserts have been placed in if-statements for performace. To
104      * enable them,
105      * set this field to true and enable them in VM with a command line
106      * flag.
107      * If you modify this class, please do test the asserts!
108      */
109     private static final boolean DEBUG = false;
110
111     /**
112      * Creates a TimSort instance to maintain the state of an ongoing

```

G:\eclair21 - GOOGLE-00-00000525\dalvik\libcore\luni\src\main\java\java\util\ComparableTimSort.java

```

    sort.
105    *
106    * @param a the array to be sorted
107    */
108    private ComparableTimSort(Object[] a) {
109        this.a = a;
110
111        // Allocate temp storage (which may be increased later if
112        // necessary)
113        int len = a.length;
114        @SuppressWarnings({"unchecked", "UnnecessaryLocalVariable"})
115        Object[] newArray = new Object[len < 2 *
116            INITIAL_TMP_STORAGE_LENGTH ?
117                len >>> 1 :
118                INITIAL_TMP_STORAGE_LENGTH];
119        tmp = newArray;
120
121        /*
122         * Allocate runs-to-be-merged stack (which cannot be expanded).
123         * The
124         * stack length requirements are described in listsort.txt. The
125         * C
126         * version always uses the same stack length (85), but this was
127         * measured to be too expensive when sorting "mid-sized" arrays
128         * (e.g.,
129         * 100 elements) in Java. Therefore, we use smaller (but
130         * sufficiently
131         * large) stack lengths for smaller arrays. The "magic numbers"
132         * in the
133         * computation below must be changed if MIN_MERGE is decreased.
134         * See
135         * the MIN_MERGE declaration above for more information.
136         */
137        int stackLen = (len < 120 ? 5 :
138            len < 1542 ? 10 :
139            len < 119151 ? 19 : 40);
140        runBase = new int[stackLen];
141        runLen = new int[stackLen];
142    }
143
144    /*
145     * The next two methods (which are package private and static)
146     * constitute
147     * the entire API of this class. Each of these methods obeys the
148     * contract
149     * of the public method with the same signature in java.util.Arrays.
150     */
151
152    static void sort(Object[] a) {
153        sort(a, 0, a.length);
154    }
155
156    static void sort(Object[] a, int lo, int hi) {
157        rangeCheck(a.length, lo, hi);
158        int nRemaining = hi - lo;
159        if (nRemaining < 2)
160            return; // Arrays of size 0 and 1 are always sorted
161
162        // If array is small, do a "mini-TimSort" with no merges
163        if (nRemaining < MIN_MERGE) {

```

G:\eclair21 - GOOGLE-00-00000525\dalvik\libcore\luni\src\main\java\java\util\ComparableTimSort.java

```

153         int initRunLen = countRunAndMakeAscending(a, lo, hi);
154         binarySort(a, lo, hi, lo + initRunLen);
155         return;
156     }
157
158     /**
159     * March over the array once, left to right, finding natural
160     * runs,
161     * extending short natural runs to minRun elements, and merging
162     * runs
163     * to maintain stack invariant.
164     */
165     ComparableTimSort ts = new ComparableTimSort(a);
166     int minRun = minRunLength(nRemaining);
167     do {
168         // Identify next run
169         int runLen = countRunAndMakeAscending(a, lo, hi);
170
171         // If run is short, extend to min(minRun, nRemaining)
172         if (runLen < minRun) {
173             int force = nRemaining <= minRun ? nRemaining : minRun;
174             binarySort(a, lo, lo + force, lo + runLen);
175             runLen = force;
176         }
177
178         // Push run onto pending-run stack, and maybe merge
179         ts.pushRun(lo, runLen);
180         ts.mergeCollapse();
181
182         // Advance to find next run
183         lo += runLen;
184         nRemaining -= runLen;
185     } while (nRemaining != 0);
186
187     // Merge all remaining runs to complete sort
188     if (DEBUG) assert lo == hi;
189     ts.mergeForceCollapse();
190     if (DEBUG) assert ts.stackSize == 1;
191 }
192
193 /**
194 * Sorts the specified portion of the specified array using a binary
195 * insertion sort. This is the best method for sorting small
196 * numbers
197 * of elements. It requires O(n log n) compares, but O(n^2) data
198 * movement (worst case).
199 *
200 * If the initial part of the specified range is already sorted,
201 * this method can take advantage of it: the method assumes that the
202 * elements from index {@code lo}, inclusive, to {@code start},
203 * exclusive are already sorted.
204 *
205 * @param a the array in which a range is to be sorted
206 * @param lo the index of the first element in the range to be
207 * sorted
208 * @param hi the index after the last element in the range to be
209 * sorted
210 * @param start the index of the first element in the range that is
211 * not already known to be sorted (@code lo <= start <= hi)
212 */

```

G:\eclair21 - GOOGLE-00-00000525\dalvik\libcore\luni\src\main\java\java\util\ComparableTimSort.java

```

208     @SuppressWarnings("fallthrough")
209     private static void binarySort(Object[] a, int lo, int hi, int
start) {
210         if (DEBUG) assert lo <= start && start <= hi;
211         if (start == lo)
212             start++;
213         for ( ; start < hi; start++) {
214             @SuppressWarnings("unchecked")
215             Comparable<Object> pivot = (Comparable) a[start];
216
217             // Set left (and right) to the index where a[start] (pivot)
belongs
218             int left = lo;
219             int right = start;
220             if (DEBUG) assert left <= right;
221             /*
222              * Invariants:
223              *   pivot >= all in [lo, left).
224              *   pivot <  all in [right, start).
225              */
226             while (left < right) {
227                 int mid = (left + right) >>> 1;
228                 if (pivot.compareTo(a[mid]) < 0)
229                     right = mid;
230                 else
231                     left = mid + 1;
232             }
233             if (DEBUG) assert left == right;
234
235             /*
236              * The invariants still hold: pivot >= all in [lo, left) and
237              * pivot < all in [left, start), so pivot belongs at left.
238              Note
239              * that if there are elements equal to pivot, left points to
the
240              * first slot after them -- that's why this sort is stable.
241              * Slide elements over to make room for pivot.
242              */
242             int n = start - left; // The number of elements to move
243             // Switch is just an optimization for arraycopy in default
case
244             switch(n) {
245                 case 2: a[left + 2] = a[left + 1];
246                 case 1: a[left + 1] = a[left];
247                     break;
248                 default: System.arraycopy(a, left, a, left + 1, n);
249             }
250             a[left] = pivot;
251         }
252     }
253
254     /**
255     * Returns the length of the run beginning at the specified position
in
256     * the specified array and reverses the run if it is descending
(ensuring
257     * that the run will always be ascending when the method returns).
258     *
259     * A run is the longest ascending sequence with:
260     *

```

G:\eclair21 - GOOGLE-00-00000525\dalvik\libcore\luni\src\main\java\java\util\ComparableTimSort.java

```

261      *      a[lo] <= a[lo + 1] <= a[lo + 2] <= ...
262      *
263      * or the longest descending sequence with:
264      *
265      *      a[lo] > a[lo + 1] > a[lo + 2] > ...
266      *
267      * For its intended use in a stable mergesort, the strictness of the
268      * definition of "descending" is needed so that the call can safely
269      * reverse a descending sequence without violating stability.
270      *
271      * @param a the array in which a run is to be counted and possibly
272      * reversed
273      * @param lo index of the first element in the run
274      * @param hi index after the last element that may be contained in
275      * the run.
276      *      It is required that @code{lo < hi}.
277      * @return the length of the run beginning at the specified
278      * position in
279      * the specified array
280      */
281 @SuppressWarnings("unchecked")
282 private static int countRunAndMakeAscending(Object[] a, int lo, int
283 hi) {
284     if (DEBUG) assert lo < hi;
285     int runHi = lo + 1;
286     if (runHi == hi)
287         return 1;
288
289     // Find end of run, and reverse range if descending
290     if (((Comparable) a[runHi++]).compareTo(a[lo]) < 0) { //
291         Descending
292         while(runHi < hi && ((Comparable)
293             a[runHi]).compareTo(a[runHi - 1]) < 0)
294             runHi++;
295         reverseRange(a, lo, runHi);
296     } else {
297         // Ascending
298         while (runHi < hi && ((Comparable)
299             a[runHi]).compareTo(a[runHi - 1]) >= 0)
300             runHi++;
301     }
302     return runHi - lo;
303 }
304
305 /**
306  * Reverse the specified range of the specified array.
307  *
308  * @param a the array in which a range is to be reversed
309  * @param lo the index of the first element in the range to be
310  * reversed
311  * @param hi the index after the last element in the range to be
312  * reversed
313  */
314 private static void reverseRange(Object[] a, int lo, int hi) {
315     hi--;
316     while (lo < hi) {
317         Object t = a[lo];
318         a[lo++] = a[hi];
319         a[hi--] = t;
320     }
321 }

```


G:\eclair21 - GOOGLE-00-00000525\dalvik\libcore\luni\src\main\java\java\util\ComparableTimSort.java

```

312     }
313
314     /**
315      * Returns the minimum acceptable run length for an array of the
316      * specified
317      * length. Natural runs shorter than this will be extended with
318      * {@link #binarySort}.
319      *
320      * Roughly speaking, the computation is:
321      *
322      *   If n < MIN_MERGE, return n (it's too small to bother with fancy
323      *   stuff).
324      *   Else if n is an exact power of 2, return MIN_MERGE/2.
325      *   Else return an int k, MIN_MERGE/2 <= k <= MIN_MERGE, such that
326      *   n/k
327      *   is close to, but strictly less than, an exact power of 2.
328      *
329      * For the rationale, see listsort.txt.
330      *
331      * @param n the length of the array to be sorted
332      * @return the length of the minimum run to be merged
333      */
334     private static int minRunLength(int n) {
335         if (DEBUG) assert n >= 0;
336         int r = 0;          // Becomes 1 if any 1 bits are shifted off
337         while (n >= MIN_MERGE) {
338             r |= (n & 1);
339             n >>= 1;
340         }
341         return n + r;
342     }
343
344     /**
345      * Pushes the specified run onto the pending-run stack.
346      *
347      * @param runBase index of the first element in the run
348      * @param runLen  the number of elements in the run
349      */
350     private void pushRun(int runBase, int runLen) {
351         this.runBase[stackSize] = runBase;
352         this.runLen[stackSize] = runLen;
353         stackSize++;
354     }
355
356     /**
357      * Examines the stack of runs waiting to be merged and merges
358      * adjacent runs
359      * until the stack invariants are reestablished:
360      *
361      *   1. runLen[i - 3] > runLen[i - 2] + runLen[i - 1]
362      *   2. runLen[i - 2] > runLen[i - 1]
363      *
364      * This method is called each time a new run is pushed onto the
365      * stack,
366      * so the invariants are guaranteed to hold for i < stackSize upon
367      * entry to the method.
368      */
369     private void mergeCollapse() {
370         while (stackSize > 1) {
371             int n = stackSize - 2;

```

G:\eclair21 - GOOGLE-00-00000525\dalvik\libcore\luni\src\main\java\java\util\ComparableTimSort.java

```

367         if (n > 0 && runLen[n-1] <= runLen[n] + runLen[n+1]) {
368             if (runLen[n - 1] < runLen[n + 1])
369                 n--;
370             mergeAt(n);
371         } else if (runLen[n] <= runLen[n + 1]) {
372             mergeAt(n);
373         } else {
374             break; // Invariant is established
375         }
376     }
377 }
378
379 /**
380  * Merges all runs on the stack until only one remains. This method
381  * is
382  * called once, to complete the sort.
383  */
384 private void mergeForceCollapse() {
385     while (stackSize > 1) {
386         int n = stackSize - 2;
387         if (n > 0 && runLen[n - 1] < runLen[n + 1])
388             n--;
389         mergeAt(n);
390     }
391 }
392
393 /**
394  * Merges the two runs at stack indices i and i+1. Run i must be
395  * the penultimate or antepenultimate run on the stack. In other
396  * words,
397  * i must be equal to stackSize-2 or stackSize-3.
398  *
399  * @param i stack index of the first of the two runs to merge
400  */
401 @SuppressWarnings("unchecked")
402 private void mergeAt(int i) {
403     if (DEBUG) assert stackSize >= 2;
404     if (DEBUG) assert i >= 0;
405     if (DEBUG) assert i == stackSize - 2 || i == stackSize - 3;
406
407     int base1 = runBase[i];
408     int len1 = runLen[i];
409     int base2 = runBase[i + 1];
410     int len2 = runLen[i + 1];
411     if (DEBUG) assert len1 > 0 && len2 > 0;
412     if (DEBUG) assert base1 + len1 == base2;
413
414     /**
415      * Record the length of the combined runs; if i is the 3rd-last
416      * run now, also slide over the last run (which isn't involved
417      * in this merge). The current run (i+1) goes away in any case.
418      */
419     runLen[i] = len1 + len2;
420     if (i == stackSize - 3) {
421         runBase[i + 1] = runBase[i + 2];
422         runLen[i + 1] = runLen[i + 2];
423     }
424     stackSize--;
425 }

```

G:\eclair21 - GOOGLE-00-00000525\dalvik\libcore\luni\src\main\java\java\util\ComparableTimSort.java

```

425         * Find where the first element of run2 goes in run1. Prior
         elements
426         * in run1 can be ignored (because they're already in place).
427         */
428         int k = gallopRight((Comparable<Object>) a[base2], a, base1,
         len1, 0);
429         if (DEBUG) assert k >= 0;
430         base1 += k;
431         len1 -= k;
432         if (len1 == 0)
433             return;
434
435         /*
436         * Find where the last element of run1 goes in run2. Subsequent
         elements
437         * in run2 can be ignored (because they're already in place).
438         */
439         len2 = gallopLeft((Comparable<Object>) a[base1 + len1 - 1], a,
         base2, len2, len2 - 1);
440         if (DEBUG) assert len2 >= 0;
441         if (len2 == 0)
442             return;
443
444         // Merge remaining runs, using tmp array with min(len1, len2)
         elements
446         if (len1 <= len2)
447             mergeLo(base1, len1, base2, len2);
448         else
449             mergeHi(base1, len1, base2, len2);
450     }
451
452     /**
453     * Locates the position at which to insert the specified key into
         the
454     * specified sorted range; if the range contains an element equal to
         key,
455     * returns the index of the leftmost equal element.
456     *
457     * @param key the key whose insertion point to search for
458     * @param a the array in which to search
459     * @param base the index of the first element in the range
460     * @param len the length of the range; must be > 0
461     * @param hint the index at which to begin the search, 0 <= hint <
         n.
462     *     The closer hint is to the result, the faster this method will
         run.
463     * @return the int k, 0 <= k <= n such that a[b + k - 1] < key <=
         a[b + k],
464     *     pretending that a[b - 1] is minus infinity and a[b + n] is
         infinity.
465     *     In other words, key belongs at index b + k; or in other words,
466     *     the first k elements of a should precede key, and the last n -
         k
467     *     should follow it.
468     */
469     private static int gallopLeft(Comparable<Object> key, Object[] a,
         int base, int len, int hint) {
470         if (DEBUG) assert len > 0 && hint >= 0 && hint < len;
471
472         int lastOfs = 0;

```

G:\eclair21 - GOOGLE-00-00000525\dalvik\libcore\luni\src\main\java\java\util\ComparableTimSort.java

```

474         int ofs = 1;
475         if (key.compareTo(a[base + hint]) > 0) {
476             // Gallop right until a[base+hint+lastOfs] < key <=
             a[base+hint+ofs]
477             int maxOfs = len - hint;
478             while (ofs < maxOfs && key.compareTo(a[base + hint + ofs]) >
                0) {
479                 lastOfs = ofs;
480                 ofs = (ofs << 1) + 1;
481                 if (ofs <= 0) // int overflow
482                     ofs = maxOfs;
483             }
484             if (ofs > maxOfs)
485                 ofs = maxOfs;
486
487             // Make offsets relative to base
488             lastOfs += hint;
489             ofs += hint;
490         } else { // key <= a[base + hint]
491             // Gallop left until a[base+hint-ofs] < key <=
             a[base+hint-lastOfs]
492             final int maxOfs = hint + 1;
493             while (ofs < maxOfs && key.compareTo(a[base + hint - ofs])
                <= 0) {
494                 lastOfs = ofs;
495                 ofs = (ofs << 1) + 1;
496                 if (ofs <= 0) // int overflow
497                     ofs = maxOfs;
498             }
499             if (ofs > maxOfs)
500                 ofs = maxOfs;
501
502             // Make offsets relative to base
503             int tmp = lastOfs;
504             lastOfs = hint - ofs;
505             ofs = hint - tmp;
506         }
507         if (DEBUG) assert -1 <= lastOfs && lastOfs < ofs && ofs <= len;
508
509         /*
510          * Now a[base+lastOfs] < key <= a[base+ofs], so key belongs
             somewhere
511          * to the right of lastOfs but no farther right than ofs. Do a
             binary
512          * search, with invariant a[base + lastOfs - 1] < key <= a[base
             + ofs].
513          */
514         lastOfs++;
515         while (lastOfs < ofs) {
516             int m = lastOfs + ((ofs - lastOfs) >>> 1);
517
518             if (key.compareTo(a[base + m]) > 0)
519                 lastOfs = m + 1; // a[base + m] < key
520             else
521                 ofs = m; // key <= a[base + m]
522         }
523         if (DEBUG) assert lastOfs == ofs; // so a[base + ofs - 1] <
             key <= a[base + ofs]
524         return ofs;
525     }

```

G:\eclair21 - GOOGLE-00-00000525\dalvik\libcore\luni\src\main\java\java\util\ComparableTimSort.java

```

526
527  /**
528   * Like gallopLeft, except that if the range contains an element
529   * equal to
530   * key, gallopRight returns the index after the rightmost equal
531   * element.
532   *
533   * @param key the key whose insertion point to search for
534   * @param a the array in which to search
535   * @param base the index of the first element in the range
536   * @param len the length of the range; must be > 0
537   * @param hint the index at which to begin the search, 0 <= hint <
538   * n.
539   * The closer hint is to the result, the faster this method will
540   * run.
541   * @return the int k, 0 <= k <= n such that a[b + k - 1] <= key <
542   * a[b + k]
543   */
544 private static int gallopRight(Comparable<Object> key, Object[] a,
545   int base, int len, int hint) {
546   if (DEBUG) assert len > 0 && hint >= 0 && hint < len;
547
548   int ofs = 1;
549   int lastOfs = 0;
550   if (key.compareTo(a[base + hint]) < 0) {
551     // Gallop left until a[b+hint - ofs] <= key < a[b+hint -
552     lastOfs]
553     int maxOfs = hint + 1;
554     while (ofs < maxOfs && key.compareTo(a[base + hint - ofs]) <
555     0) {
556       lastOfs = ofs;
557       ofs = (ofs << 1) + 1;
558       if (ofs <= 0) // int overflow
559         ofs = maxOfs;
560     }
561     if (ofs > maxOfs)
562       ofs = maxOfs;
563
564     // Make offsets relative to b
565     int tmp = lastOfs;
566     lastOfs = hint - ofs;
567     ofs = hint - tmp;
568   } else { // a[b + hint] <= key
569     // Gallop right until a[b+hint + lastOfs] <= key < a[b+hint
570     + ofs]
571     int maxOfs = len - hint;
572     while (ofs < maxOfs && key.compareTo(a[base + hint + ofs])
573     >= 0) {
574       lastOfs = ofs;
575       ofs = (ofs << 1) + 1;
576       if (ofs <= 0) // int overflow
577         ofs = maxOfs;
578     }
579     if (ofs > maxOfs)
580       ofs = maxOfs;
581
582     // Make offsets relative to b
583     lastOfs += hint;
584     ofs += hint;
585   }
586 }

```

G:\eclair21 - GOOGLE-00-00000525\dalvik\libcore\luni\src\main\java\java\util\ComparableTimSort.java

```

577         if (DEBUG) assert -1 <= lastOfs && lastOfs < ofs && ofs <= len;
578
579         /*
580          * Now a[b + lastOfs] <= key < a[b + ofs], so key belongs
581          * somewhere to
582          * the right of lastOfs but no farther right than ofs. Do a
583          * binary
584          * search, with invariant a[b + lastOfs - 1] <= key < a[b +
585          * ofs].
586          */
587         lastOfs++;
588         while (lastOfs < ofs) {
589             int m = lastOfs + ((ofs - lastOfs) >>> 1);
590
591             if (key.compareTo(a[base + m]) < 0)
592                 ofs = m;           // key < a[b + m]
593             else
594                 lastOfs = m + 1;   // a[b + m] <= key
595         }
596         if (DEBUG) assert lastOfs == ofs;    // so a[b + ofs - 1] <= key
597         < a[b + ofs]
598         return ofs;
599     }
600
601     /**
602     * Merges two adjacent runs in place, in a stable fashion. The
603     * first
604     * element of the first run must be greater than the first element
605     * of the
606     * second run (a[base1] > a[base2]), and the last element of the
607     * first run
608     * (a[base1 + len1 - 1]) must be greater than all elements of the
609     * second run.
610     *
611     * For performance, this method should be called only when len1 <=
612     * len2;
613     * its twin, mergeHi should be called if len1 >= len2. (Either
614     * method
615     * may be called if len1 == len2.)
616     *
617     * @param base1 index of first element in first run to be merged
618     * @param len1  length of first run to be merged (must be > 0)
619     * @param base2 index of first element in second run to be merged
620     *             (must be aBase + aLen)
621     * @param len2  length of second run to be merged (must be > 0)
622     */
623     @SuppressWarnings("unchecked")
624     private void mergeLo(int base1, int len1, int base2, int len2) {
625         if (DEBUG) assert len1 > 0 && len2 > 0 && base1 + len1 == base2;
626
627         // Copy first run into temp array
628         Object[] a = this.a; // For performance
629         Object[] tmp = ensureCapacity(len1);
630         System.arraycopy(a, base1, tmp, 0, len1);
631
632         int cursor1 = 0;           // Indexes into tmp array
633         int cursor2 = base2;       // Indexes into a
634         int dest = base1;          // Indexes into a
635
636         // Move first element of second run and deal with degenerate

```

G:\eclair21 - GOOGLE-00-00000525\dalvik\libcore\luni\src\main\java\java\util\ComparableTimSort.java

```

        cases
627     a[dest++] = a[cursor2++];
628     if (--len2 == 0) {
629         System.arraycopy(tmp, cursor1, a, dest, len1);
630         return;
631     }
632     if (len1 == 1) {
633         System.arraycopy(a, cursor2, a, dest, len2);
634         a[dest + len2] = tmp[cursor1]; // Last elt of run 1 to end
        of merge
635         return;
636     }
637
638     int minGallop = this.minGallop; // Use local variable for
        performance
639     outer:
640     while (true) {
641         int count1 = 0; // Number of times in a row that first run
        won
642         int count2 = 0; // Number of times in a row that second run
        won
643
644         /*
645          * Do the straightforward thing until (if ever) one run
        starts
646          * winning consistently.
647          */
648         do {
649             if (DEBUG) assert len1 > 1 && len2 > 0;
650             if (((Comparable) a[cursor2]).compareTo(tmp[cursor1]) <
        0) {
651                 a[dest++] = a[cursor2++];
652                 count2++;
653                 count1 = 0;
654                 if (--len2 == 0)
655                     break outer;
656             } else {
657                 a[dest++] = tmp[cursor1++];
658                 count1++;
659                 count2 = 0;
660                 if (--len1 == 1)
661                     break outer;
662             }
663         } while ((count1 | count2) < minGallop);
664
665         /*
666          * One run is winning so consistently that galloping may be
        a
667          * huge win. So try that, and continue galloping until (if
        ever)
668          * neither run appears to be winning consistently anymore.
669          */
670         do {
671             if (DEBUG) assert len1 > 1 && len2 > 0;
672             count1 = gallopRight((Comparable) a[cursor2], tmp,
        cursor1, len1, 0);
673             if (count1 != 0) {
674                 System.arraycopy(tmp, cursor1, a, dest, count1);
675                 dest += count1;
676                 cursor1 += count1;

```

G:\eclair21 - GOOGLE-00-00000525\dalvik\libcore\luni\src\main\java\java\util\ComparableTimSort.java

```

677             len1 -= count1;
678             if (len1 <= 1) // len1 == 1 || len1 == 0
679                 break outer;
680         }
681         a[dest++] = a[cursor2++];
682         if (--len2 == 0)
683             break outer;
684
685         count2 = gallopLeft((Comparable) tmp[cursor1], a,
686                             cursor2, len2, 0);
687         if (count2 != 0) {
688             System.arraycopy(a, cursor2, a, dest, count2);
689             dest += count2;
690             cursor2 += count2;
691             len2 -= count2;
692             if (len2 == 0)
693                 break outer;
694         }
695         a[dest++] = tmp[cursor1++];
696         if (--len1 == 1)
697             break outer;
698         minGallop--;
699         } while (count1 >= MIN_GALLOP | count2 >= MIN_GALLOP);
700         if (minGallop < 0)
701             minGallop = 0;
702         minGallop += 2; // Penalize for leaving gallop mode
703     } // End of "outer" loop
704     this.minGallop = minGallop < 1 ? 1 : minGallop; // Write back
705     to field
706
707     if (len1 == 1) {
708         if (DEBUG) assert len2 > 0;
709         System.arraycopy(a, cursor2, a, dest, len2);
710         a[dest + len2] = tmp[cursor1]; // Last elt of run 1 to end
711         of merge
712     } else if (len1 == 0) {
713         throw new IllegalArgumentException(
714             "Comparison method violates its general contract!");
715     } else {
716         if (DEBUG) assert len2 == 0;
717         if (DEBUG) assert len1 > 1;
718         System.arraycopy(tmp, cursor1, a, dest, len1);
719     }
720 }
721
722 /**
723  * Like mergeLo, except that this method should be called only if
724  * len1 >= len2; mergeLo should be called if len1 <= len2. (Either
725  * method
726  * may be called if len1 == len2.)
727  *
728  * @param base1 index of first element in first run to be merged
729  * @param len1  length of first run to be merged (must be > 0)
730  * @param base2 index of first element in second run to be merged
731  *              (must be aBase + aLen)
732  * @param len2  length of second run to be merged (must be > 0)
733  */
734 @SuppressWarnings("unchecked")
735 private void mergeHi(int base1, int len1, int base2, int len2) {
736     if (DEBUG) assert len1 > 0 && len2 > 0 && base1 + len1 == base2;

```


G:\eclair21 - GOOGLE-00-00000525\dalvik\libcore\luni\src\main\java\java\util\ComparableTimSort.java

```

733
734     // Copy second run into temp array
735     Object[] a = this.a; // For performance
736     Object[] tmp = ensureCapacity(len2);
737     System.arraycopy(a, base2, tmp, 0, len2);
738
739     int cursor1 = base1 + len1 - 1; // Indexes into a
740     int cursor2 = len2 - 1;        // Indexes into tmp array
741     int dest = base2 + len2 - 1;    // Indexes into a
742
743     // Move last element of first run and deal with degenerate cases
744     a[dest--] = a[cursor1--];
745     if (--len1 == 0) {
746         System.arraycopy(tmp, 0, a, dest - (len2 - 1), len2);
747         return;
748     }
749     if (len2 == 1) {
750         dest -= len1;
751         cursor1 -= len1;
752         System.arraycopy(a, cursor1 + 1, a, dest + 1, len1);
753         a[dest] = tmp[cursor2];
754         return;
755     }
756
757     int minGallop = this.minGallop; // Use local variable for
    performance
758 outer:
759     while (true) {
760         int count1 = 0; // Number of times in a row that first run
    won
761         int count2 = 0; // Number of times in a row that second run
    won
762
763         /*
764          * Do the straightforward thing until (if ever) one run
765          * appears to win consistently.
766          */
767         do {
768             if (DEBUG) assert len1 > 0 && len2 > 1;
769             if (((Comparable) tmp[cursor2]).compareTo(a[cursor1]) <
    0) {
770                 a[dest--] = a[cursor1--];
771                 count1++;
772                 count2 = 0;
773                 if (--len1 == 0)
774                     break outer;
775             } else {
776                 a[dest--] = tmp[cursor2--];
777                 count2++;
778                 count1 = 0;
779                 if (--len2 == 1)
780                     break outer;
781             }
782         } while ((count1 | count2) < minGallop);
783
784         /*
785          * One run is winning so consistently that galloping may be
    a
786          * huge win. So try that, and continue galloping until (if
    ever)

```

G:\eclair21 - GOOGLE-00-00000525\dalvik\libcore\luni\src\main\java\java\util\ComparableTimSort.java

```

787         * neither run appears to be winning consistently anymore.
788         */
789         do {
790             if (DEBUG) assert len1 > 0 && len2 > 1;
791             count1 = len1 - gallopRight((Comparable) tmp[cursor2],
792                                     a, base1, len1, len1 - 1);
793             if (count1 != 0) {
794                 dest -= count1;
795                 cursor1 -= count1;
796                 len1 -= count1;
797                 System.arraycopy(a, cursor1 + 1, a, dest + 1,
798                                 count1);
799                 if (len1 == 0)
800                     break outer;
801             }
802             a[dest--] = tmp[cursor2--];
803             if (--len2 == 1)
804                 break outer;
805
806             count2 = len2 - gallopLeft((Comparable) a[cursor1], tmp,
807                                     0, len2, len2 - 1);
808             if (count2 != 0) {
809                 dest -= count2;
810                 cursor2 -= count2;
811                 len2 -= count2;
812                 System.arraycopy(tmp, cursor2 + 1, a, dest + 1,
813                                 count2);
814                 if (len2 <= 1)
815                     break outer; // len2 == 1 || len2 == 0
816             }
817             a[dest--] = a[cursor1--];
818             if (--len1 == 0)
819                 break outer;
820             minGallop--;
821             } while (count1 >= MIN_GALLOP | count2 >= MIN_GALLOP);
822             if (minGallop < 0)
823                 minGallop = 0;
824             minGallop += 2; // Penalize for leaving gallop mode
825         } // End of "outer" loop
826         this.minGallop = minGallop < 1 ? 1 : minGallop; // Write back
827         to field
828
829         if (len2 == 1) {
830             if (DEBUG) assert len1 > 0;
831             dest -= len1;
832             cursor1 -= len1;
833             System.arraycopy(a, cursor1 + 1, a, dest + 1, len1);
834             a[dest] = tmp[cursor2]; // Move first elt of run2 to front
835             of merge
836         } else if (len2 == 0) {
837             throw new IllegalArgumentException(
838                 "Comparison method violates its general contract!");
839         } else {
840             if (DEBUG) assert len1 == 0;
841             if (DEBUG) assert len2 > 0;
842             System.arraycopy(tmp, 0, a, dest - (len2 - 1), len2);
843         }
844     }
845 }
846 /**

```

G:\eclair21 - GOOGLE-00-00000525\dalvik\libcore\luni\src\main\java\java\util\ComparableTimSort.java

```

841     * Ensures that the external array tmp has at least the specified
842     * number of elements, increasing its size if necessary. The size
843     * increases exponentially to ensure amortized linear time
      complexity.
844     *
845     * @param minCapacity the minimum required capacity of the tmp array
846     * @return tmp, whether or not it grew
847     */
848     private Object[] ensureCapacity(int minCapacity) {
849         if (tmp.length < minCapacity) {
850             // Compute smallest power of 2 > minCapacity
851             int newSize = minCapacity;
852             newSize |= newSize >> 1;
853             newSize |= newSize >> 2;
854             newSize |= newSize >> 4;
855             newSize |= newSize >> 8;
856             newSize |= newSize >> 16;
857             newSize++;
858
859             if (newSize < 0) // Not bloody likely!
860                 newSize = minCapacity;
861             else
862                 newSize = Math.min(newSize, a.length >>> 1);
863
864             @SuppressWarnings({"unchecked", "UnnecessaryLocalVariable"})
865             Object[] newArray = new Object[newSize];
866             tmp = newArray;
867         }
868         return tmp;
869     }
870
871     /**
872     * Checks that fromIndex and toIndex are in range, and throws an
873     * appropriate exception if they aren't.
874     *
875     * @param arrayLen the length of the array
876     * @param fromIndex the index of the first element of the range
877     * @param toIndex the index after the last element of the range
878     * @throws IllegalArgumentException if fromIndex > toIndex
879     * @throws ArrayIndexOutOfBoundsException if fromIndex < 0
880     *         or toIndex > arrayLen
881     */
882     private static void rangeCheck(int arrayLen, int fromIndex, int
      toIndex) {
883         if (fromIndex > toIndex)
884             throw new IllegalArgumentException("fromIndex(" + fromIndex
885                 + " > toIndex(" + toIndex + ")");
886         if (fromIndex < 0)
887             throw new ArrayIndexOutOfBoundsException(fromIndex);
888         if (toIndex > arrayLen)
889             throw new ArrayIndexOutOfBoundsException(toIndex);
890     }
891 }

```